MFIBlocks: An Effective Blocking Algorithm for Entity Resolution

Batya Kenig, Avigdor Gal

Technion-Israel Institute of Technology, Haifa, Israel

Abstract

Entity resolution is the process of discovering groups of tuples that correspond to the same real-world entity. Blocking algorithms separate tuples into blocks that are likely to contain matching pairs. Tuning is a major challenge in the blocking process and in particular, high expertise is needed in contemporary blocking algorithms to construct a *blocking key*, based on which tuples are assigned to blocks. In this work, we introduce a blocking approach that avoids selecting a blocking key altogether, relieving the user from this difficult task. The approach is based on maximal frequent itemsets selection, allowing early evaluation of block quality based on the overall commonality of its members. A unique feature of the proposed algorithm is the use of prior knowledge of the estimated size of duplicate sets in enhancing the blocking accuracy. We report on a thorough empirical analysis, using common benchmarks of both real-world and synthetic datasets to exhibit the effectiveness and efficiency of our approach.

1. Introduction

Entity resolution is a fundamental problem in data integration. It refers to the problem of determining which tuples (using relational notation) resolve to the same real-world entity. At the heart of the entity resolution process is the challenge to match tuples that share no unique identifiers, may come from non-matching schemata, and may consist of typos and out-of-date or missing information. Entity resolution algorithms typically compare the content of tuples to determine if they match and merge matching tuples into one. Such a comparison may be prohibitive for big datasets if all tuple pairs are compared and hence pairwise comparison is typically preceded by a blocking phase, a procedure that divides tuples into mutually exclusive subsets called *blocks*.

Tuples assigned to the same block are ideally resolved to the same real-world entity. In practice, tuples in a block are all candidates for the more rigorous tuple pair-wise comparison. Therefore, blocking algorithms should be designed to produce quality blocks, containing as many tuple matches and avoiding as many non-matches as possible. Balancing the two requirements calls for an optimal block size that should not

Email address: {batyak@tx, avigal@ie}.technion.ac.il (Batya Kenig, Avigdor Gal)

be too small, to avoid false negatives, or too large, to avoid false positives. Larger blocks also increase the time spent on pair-wise tuple comparison and hence, blocking algorithms aim at balancing the need to reduce false negatives and the need to reduce performance overhead.

Several blocking algorithms were proposed in the literature, *e.g.*, sorted neighborhood [1], canopy clustering [2], and q-gram indexing [3]. In this work we offer a novel blocking algorithm, dubbed MFIBlocks, that is based on iteratively applying an algorithm for mining Maximal Frequent Itemsets [4]. MFIBlocks offers four major unique features. Firstly, MFIBlocks waives the need to manually design a *blocking key*, the value of one or more of a tuple's attributes. Blocking keys in contemporary blocking algorithms have to be carefully designed to avoid false negatives by assigning matching tuples to different blocks. Therefore, attributes in the blocking key should contain few errors and missing values and the design of a blocking key should take into account the frequency distribution of values in the attributes of the blocking key to balance block sizes. MFIBlocks relieves the designer from the difficult task of constructing a blocking key.

Second, MFIBlocks localizes the search for similar tuples and is able to uncover blocks of tuples that are similar in multiple, possibly overlapping sets of attributes. MFIBlocks allows a dynamic, automatic, and flexible selection of a blocking key, so that different blocks can be created based on different keys. This approach is in line with the state-of-the-art in clustering literature (see, *e.g.*, [5]) and extends the current perception of a single-key-fits-all.

Blocks, created by the algorithm, are constrained to satisfy the *compact set* (CS) and *sparse neighborhood* (SN) [6] properties. As such, local structural properties of the dataset are used during the discovery and evaluation of tuple clusters and the number of comparisons for each tuple is kept low, even though the same tuple can appear in several clusters (using multiple keys) simultaneously.

Finally, MFIBlocks is designed to discover entity sets of matching tuples with largely varying sizes. MFIBlocks effectively utilizes a-priori knowledge of the sizes of matching entity sets, by discovering clusters of the appropriate size having the largest possible commonality.

This paper introduces the MFIBlocks algorithm, discusses its properties, and presents a thorough empirical analysis, demonstrating its superior effectiveness. We offer techniques to make the execution time performance of the algorithm attractive, balancing execution time with effectiveness. The novelty of our paper is as follows:

- We present a novel blocking algorithm that reduces the effort of manual tuning and enables locating clusters of similar tuples in multiple, possibly overlapping sets of attributes.
- We provide a thorough empirical analysis of the algorithm performance, using both real-world and synthetic datasets, and show its superior effectiveness over common benchmarks.
- We offer methods to ensure the efficiency of the algorithm, demonstrating the trade-off between execution time and effectiveness.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the entity resolution process and frequent itemset mining. The building blocks of the proposed approach are provided in Section 3. The algorithm is presented in Section 4. In Section 5 we provide an empirical analysis over several benchmark datasets. Section 6 provides an overview of related work, positioning our work on this background. We conclude in Section 7 with a summary and a discussion of future work.

2. Preliminaries

The general, unsupervised entity resolution process illustrated in Figure 1 contains blocking, comparison and classification stages. Occasionally, a standardization process precedes these steps to increase the process effectiveness. The output of the blocking stage is a set of blocks where each pair of tuples in a block is considered a *candidate pair*. Only candidate pairs are then compared and other pairs are automatically classified as non-matches.

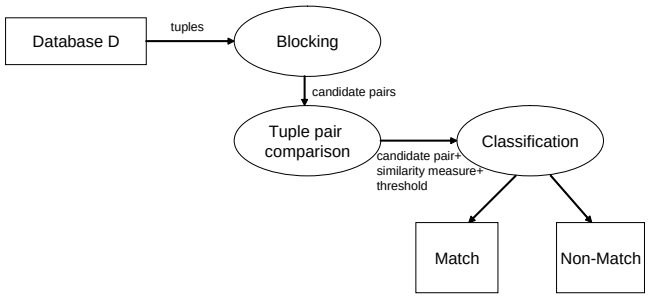


Figure 1: Entity resolution process

Using similarity measures such as the Jaccard coefficient [7], candidate pairs are classified as matching or non-matching. A tuple pair (t_1, t_2), over a schema of k comparable attributes, is represented as a vector $v = [v_1, ..., v_k]$. Each v_i is a measure of the similarity of the i-th attribute. In most cases, the entries in the vector v are in the range $[0, 1]$. A function f over the values of these entries is used to classify a pair according to a predefined threshold.

In this paper we focus on the blocking stage and suggest using maximal frequent itemsets to generate blocks of tuples as candidates for the classification stage. For completeness sake, we now provide an overview of the concepts of frequent and maximal frequent itemsets [8]. Frequent itemsets originated from the data mining field and was used in other fields as well, *e.g.*, for identifying similar Web documents [9]. This overview uses the notions of items and transactions, based on the native vocabulary of data mining. It is worth noting that the term *transaction* has a different (albeit related) meaning to the same term in the database literature, referring to a bag of items from a raw dataset, *e.g.*, billing transactions. Let $M = \{I_1, I_2, \ldots, I_m\}$ be a set of items and let $T = \left\langle T_1, T_2, \cdots, T_n \right\rangle$ be a set of transactions. A transaction $T_i = (tid, I)$ with identifier tid contains a set of items $I \subseteq M$. The *support* of some set of items $I \subseteq M$

is the set of transactions in T that contain all items in I. Each transaction possibly contains additional items.

transaction	items
t_1	a_1, b_1, c_1
t_2	a_1, b_1, c_2
t_3	a_2, b_2, c_3
t_4	a_2, b_2, c_2
t_5	a_1, b_3, c_4

Table 1: Sample Transaction Database

Example 1. *Table 1 contains five transactions with the items* $M = \{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3, c_4\}$. *Transaction* t_1 *contains items* $\{a_1, b_1, c_1\}$. *The support set of itemset* $\{a_1, b_1\}$ *is the set of transactions* $\{t_1, t_2\}$.

I is *frequent* if I's support size surpasses a minimum support threshold $minsup$. The *downward closure property* ensures that a subset of a frequent itemset is also a frequent itemset [8] and therefore a frequent itemset cannot contain an infrequent itemset.

Given a transaction set and a minimum support threshold $minsup$, the problem of finding the complete set of frequent itemsets (FI) is called the *frequent itemset mining problem.* A major challenge in mining frequent itemsets from a large or dense (with many different items) dataset, is that a too low $minsup$ thresholds generates a huge number of frequent itemsets. For example, if there is a frequent itemset of size l, then all $2^l - 1$ nonempty subsets of that itemset are also frequent due to the downward closure property. To overcome this difficulty, we introduce next *maximal frequent itemsets* (MFIs for short) [10].

A frequent itemset X is *maximal* if there does not exist a frequent itemset Y such that $Y \supset X$. Due to the downward closure property of frequent itemsets, all subsets of a maximal frequent itemset are frequent, while all supersets of such an itemset are infrequent.

Example 2. *There are seven frequent itemsets in Table 1, with* $minsup = 2$*:*

$$\{\{a_1\}, \{a_2\}, \{b_1\}, \{b_2\}, \{c_2\}, \{a_1, b_1\}, \{a_2, b_2\}\}$$

and three maximal frequent itemsets with the same support of 2*:*

$$\{\{a_1, b_1\}, \{a_2, b_2\}, \{c_2\}\}$$

3. MFI-based blocking

Maximal frequent itemsets lend themselves well to the problem of entity resolution. MFIs are sets of tuples (support set) with maximal commonality over the set of values in a database. Sets created by an MFI algorithm may be interpreted as similarity over subsets of attributes, as formalized below. The added value of such sets is that the

set of attributes is chosen automatically by the algorithm, which paths the way to a flexible key selection mechanism. Whenever the expertise of choosing a clustering key exists, MFI-based blocking can take it into account (using, *e.g.*, weights). However, MFI-based blocking can effectively produce good blocking results even in the absence of such expertise. MFI-based blocks provide, beyond blocks, useful hints as to the attributes that most affect the pairing, and as such can also serve as an exploratory tool, providing useful information for the comparison and classification stages as well.

We face three main challenges when designing an MFI-based blocking algorithm. From an engineering point of view the new algorithm should fulfill this promise and provide higher quality blocks than existing blocking algorithms. In this work we show that a careful design of the algorithm yields empirical improvement of block accuracy over existing blocking mechanisms that ranges from 25% to more than 700% in terms of F-measure.

A second challenge we face has to do with algorithm tuning. The promise of MFI-based blocking is in providing a flexible algorithm that can do even without any expertise in the design of a blocking key. However, such tuning gain should not be replaced by other, possibly harder parameters to tune. The proposed algorithm requires three parameters. The first, introduced as the MFI support size ($minsup$) can be easily derived from the task at hand, as discussed in Section 4.3. The second defines a minimum threshold of cluster similarity (min_th). The third parameter specifies the minimum strength of a cluster, called neighborhood growth and formally defined below. As it turns out, these last two parameters are strongly dependent. In particular, by setting the neighborhood growth first, min_th can be set automatically. Our empirical study shows that the neighborhood growth is most beneficial in a narrow range of values, and setting it requires little expertise.

The third and final challenge has to do with performance. The problem of counting the number of maximal frequent itemsets in a transaction set is #P-complete [11], which is no better, in the worst case, then solving the blocking problem by enumerating all possible blocks. However, several algorithms have been shown empirically to efficiently mine all maximal frequent itemsets [12, 4, 13] and their use is widespread. Still, to manage the blocking process efficiently, we have strived to optimize the performance of the proposed algorithm and validated the outcome against other leading blocking algorithms, measuring trade-off between performance and effectiveness. To start off, we use as a routine in our implementation FPMax [4], which received the best implementation award and displayed the best performance for both low and high supports in the frequent itemset mining implementations report from 2003 [14]. With FPMax, we show empirically that blocking can be done in a quadratic runtime, similar to other blocking algorithms that are based on tuple pairwise comparison. Such performance may still be prohibitive for large datasets. Therefore, in Section 4 we discuss methods to improve the coefficients of this complexity. We show empirically that the proposed algorithm is competitive with reported performance of existing blocking algorithms, making it an attractive blocking alternative.

3.1. MFI-based blocking model

We now present definitions and notations that serve us in devising the algorithm. We accompany the presentation with a case study database, given in Table 2. This

tuple ID	given name	surname	address1	suburb	postcode	state	phone
t_1	xavier	tomich	gamban square	murgon	2150		
t_2	xavier	tomich	gambandsquare	mudrgon	2151		
t_3	savannah	savannah	tuthiolzlace	dayle sford	4120	nsw	0752913455
t_4	savannah	humphreys	tuthill place	daylesford	4120	nsw	0752913455
t_5	savannah	humphreys	tuthill place	daylqogrd	4102	ndw	0752913455
t_6	savannah	humphfjfeys	tuthill place	daylesord	4219	nsw	0752913455
t_7	savanmwh	humpmhheys	tuthill place	daylesford	4105	nsw	0752913455
t_8	savanah	humphreys	tuthill klace	dayles fofrd	4120	nhs	0752913455
t_9	chelsear	hanns	mugglestoea	elwood	2119	nsw	0727375124
t_{10}	chelara	hanna	mugglestonna	elwood	2161	nsw	0727375124
t_{11}	chelsea	hanna	mugglestone	elwood	2117	nsw	0727375124
t_{12}	chelsibe	habmy	mugglestone	elwood	2117	nsw	0727375124
t_{13}	raquel	rudad	wilkinsfateet	wellington point	4054	sa	0743902126
t_{14}	annabel	burleigh	wakelin circuit	newborough	2150	nsw	02 48546447

Table 2: Case Study Database

example has five entities, three of which are represented by tuple sets of varying size (6, 4 and 2). The tuples were generated using the database generator of FEBRL [15].

Consider a relation $R(A_1, A_2, ..., A_n)$ and an instance of R, $D = \{t_1, t_2, ..., t_m\}$. Given D, an MFI blocking algorithm generates a set of blocks $B_o = \left\{B^1, B^2, ..., B^p\right\}$, where a *block* $B^i \in B_o$ is a quadruple $B^i = \left\langle P^i, A^i, E^i, S^i \right\rangle$.

P^i $(1 \leq i \leq p)$ is a subset of D so that $D = \bigcup_{i=1}^{p} P^i$ and different P^i sets may overlap. P^i is called the *support* of B^i and all tuples in P^i are considered candidates to represent the same real-world entity. We denote by M^i the bag of values that appear in each of the tuples in P^i.

Each block B^i is created based on a set of attributes $A^i \subseteq \{A_1, A_2, ..., A_n\}$. Only values that are assigned to attributes A^i are considered in assessing the participation of tuples in B^i. The itemset $E^i \subseteq M^i$ is a set of values, created using values of attributes in A^i. Each such value is common to all tuples in P^i, *i.e.*, it appears in each and every tuple in P^i. Finally, S^i is a score, assigned with B^i based on the similarity of the A^i values for tuples in P^i. The score of a block B^i is computed using a score function σ over the set of tuples in P^i. A typical example of such a score function is the Jaccard similarity measure [16] for bags, $\sigma(P^i) = \frac{|E^i|}{|M^i|}$. As a notation convention, we use σ^i to denote the use of A^i attributes in computing a score.

The Compact Set (CS) and Sparse Neighborhood (SN) criteria [6] are useful for characterizing "good" clusters. These criteria capture local structural properties of clusters, where elements in each cluster are closer to each other than to other elements, making their local neighborhood empty or sparse. We find the principles behind these criteria to be also desirable properties of blocks returned by a blocking algorithm. This is because duplicates of the same entity will usually be similar to one another but have only a small number of other tuples similar to them as well. We adapt these principles for blocking algorithms, as follows.

Definition 1. *Let $A^i \subseteq \{A_1, A_2, ..., A_n\}$ be a subset of attributes and σ^i be the score function that uses A^i. An A^i-CS is a set of tuples such that for every two tuples $\{t_1, t_2\} \subseteq A^i$-CS, $\sigma^i(\{t_1, t_2\}) > \sigma^i(\{t_1, t_3\})$ where $t_3 \notin A^i$-CS.*

Definition 1 extends the definition of [6] that uses tuple distance as a score function to a general score function, while maintaining the tuple pairwise comparison. The following proposition defines a condition for A^i-CS conformance, effectively linking the essence of MFIs with entity resolution. The proposition assumes that σ is monotonic with respect to value containment. That is, for two itemsets $E^i \supseteq E^j$, $\sigma(P^i) \geq \sigma(P^j)$. As a simple example of a monotonic σ one may consider itemset cardinality ($\sigma(E) = |E|$).

Proposition 1. *Let $B^i = \langle P^i, E^i, A^i, S^i \rangle$ be a block and let σ_i be monotonic wrt value containment. P^i forms an A^i-CS.*

Proof. Consider tuples $\{t_1, t_2\} \subseteq P^i$ and assume, by contradiction, that there is a tuple $t_3 \notin P^i$, such that $\sigma^i(\{t_1, t_3\}) \geq \sigma^i(\{t_1, t_2)\})$. Let $E_{1,2}$ denote the set of values (originating from attributes A^i) common to t_1 and t_2, and let $E_{1,3}$ denote the set of values common to t_1 and t_3. We note that since $E_{1,2}$ and $E_{1,3}$ must both contain values that appear in t_1 then either $E_{1,2} \supseteq E_{1,3}$ or $E_{1,3} \supseteq E_{1,2}$. Since σ_i is monotonic and $\sigma^i(\{t_1, t_3\}) \geq \sigma^i(\{t_1, t_2)\}$ then the former cannot hold and therefore $E_{1,3} \supseteq E_{1,2}$, but then it should be part of P^i, contradicting our assumption that $t_3 \notin P^i$. □

Definition 2. *The* neighborhood *of a tuple t is a set of tuples that share some block with t: $N(t) = \bigcup_{P^i | t \in P^i} P^i$. The* neighborhood growth (NG) *of tuple t is $|N(t)|$, the cardinality of $N(t)$.*

Higher $|N(t)|$ means a higher number of pairwise comparisons for t in the entity resolution process. Our definition of Sparse Neighborhood (SN) considers $|N(t)|$ and a minimal block size $minsup$, as follows:

Definition 3. *A set of tuples $D' \subseteq D$ is considered a* sparse neighborhood *if $|D'| = 1$ or $\max_{t \in D'} |N(t)| \leq p \cdot minsup$, where $p > 1$ is a predefined constant.*

Ultimately, our goal is to find the largest set of blocks that still satisfies the sparse neighborhood constraint. When $minsup$ is set to be the size of the expected clusters, the sparse neighborhood constraint limits the number of comparisons a tuple will incur during the entity resolution process based on this size and the parameter p, which was shown in our preliminary experiments to yield good results in the range of $[1.5, 4]$ and is easily tuned.

A tuple that, upon termination of the blocking algorithm, belongs to at least one block $B^i \in B_o$ with support size equal or greater than j ($|P^i| \geq j$), will be considered j-$covered$. A tuple t is $minsup$-$covered$ if it belongs to a block $B^i \in B_o$ that is discovered by an MFI algorithm with support threshold of $minsup$. Tuples that do not belong to any $B^i \in B_o$ do not to have a match and remain singletons.

Example 3. *Table 3 presents blocks of the case study (Table 2) that were created by the algorithm (see Section 4) with $minsup = 3$. For exposition reasons, we look for clusters of size three in the dataset. Section 4.3 details how the $minsup$ parameter should be selected. Each block may use a different attribute set as a blocking key, determined automatically by the algorithm. For example, B^1 uses $A^1 = \{$given name, address1, and phone$\}$ while B^2 uses $A^2 = \{$surname, phone$\}$.*

Block B^i	Support P^i	Itemset E^i
B^1	$\{t_4, t_5, t_6\}$	{savannah, tuthill place, 0752913455}
B^2	$\{t_4, t_5, t_8\}$	{humphreys, 0752913455}
B^3	$\{t_3, t_4, t_8\}$	{4120, 0752913455}
B^4	$\{t_3, t_4, t_6, t_7\}$	{nsw, 0752913455}
B^5	$\{t_4, t_5, t_7\}$	{0752913455, 3050536}
B^6	$\{t_9, t_{10}, t_{11}\}$	{elwood, nsw, 19920405, 0727375124}
B^7	$\{t_{10}, t_{11}, t_{12}\}$	{nsw, 21, 0727375124}

Table 3: Blocks of the Case Study

Using the Jaccard similarity measure as defined above, we arrive at $\sigma(P^1) = \frac{3}{14}$ *and* $\sigma(P^6) = \frac{4}{17}$*. The blocks in Table 3 are all* A_i*-*CS*. For example, there is no tuple closer to* t_4 *over* $A^1 = \{$*given name, address1, phone*$\}$ *than* t_5 *and* t_6*. Likewise, there is no tuple closer to* t_4 *over* $A^3 = \{$*postcode, phone*$\}$ *than* t_3 *and* t_8*.*

Also, for $p = 2$*, the generated block set satisfies the sparse neighborhood constraint:*

$$\begin{aligned} max_{t \in D}|N(t)| &= |N(t_4)| \\ &= |P^1 \cup P^2 \cup P^3 \cup P^4 \cup P^5| \\ &= |\{t_5, t_6\} \cup \{t_5, t_8\} \cup \{t_3, t_8\} \cup \{t_3, t_6, t_7\} \cup \{t_5, t_7\}| \\ &= 5 \leq 2 \cdot minsup = 6 \end{aligned}$$

Finally, the tuples t_3*-*t_{12} *are* 3*-*$covered$*, while each of the tuples* t_1, t_2, t_{13}, t_{14} *remains a singleton. It is worth noting that the outcome of executing the algorithm with* $minsup = 3$ *does not yield the best possible set of blocks. For example, tuples* t_3*-*t_8*, representing a single entity, never appear in a single block. In Section 4.3 we show how to select the correct* $minsup$ *level, and Example 5 illustrates how the correct blocks are created.*

4. The MFIBlocks algorithm

This section introduces MFIBlocks, a blocking algorithm using maximal frequent itemsets. The algorithm is presented in a modular fashion. We start with describing the data setup (Section 4.1) followed by a method for assessing cluster quality in Section 4.2. We then discuss the selection of $minsup$ (Section 4.3) and the application of the compact set and sparse neighborhood criteria (Section 4.4). The algorithm pseudo-code is presented in Section 4.5, followed by a discussion about performance (Section 4.6). To illustrate the various elements of the algorithm, we use Table 2.

At a high-level, the algorithm operates as follows: after a preparatory phase, a series of iterations proceeds. In each iteration, the MFI mining algorithm is run on a decreasing set of uncovered tuples, creating a new set of blocks. After each iteration, the blocks are scored. The final set of blocks returned as output consists of the largest possible set of high-scoring blocks such that the sparse neighborhood criterion is met.

4.1. Data setup

Frequent itemset mining algorithms run on a dataset of *transactions*, where each transaction is composed of a set of items I. In this preparatory phase we transform a database into a suitable input dataset for the MFI algorithm. We would like to provide support for overcoming variations that stem from abbreviations, synonyms, and typographical errors and also to maintain information regarding the structural information of the data. Therefore, as a first step, a lexicon of items is created and enumerated. Each attribute value is separated into q-grams [17], a known approach in Information Retrieval for overcoming typographical errors. Identical q-grams, originating from the same attribute, receive the same id while identical q-grams that originate from different attributes are assigned different ids. Once the lexicon is constructed, database tuples are transformed into a transaction dataset by replacing attribute values with the ids of their corresponding q-grams. At this point, a tuple t becomes a set of items, referred to as $t.items$, where each item id, $id \in t.items$ corresponds to a q-gram in t. Since an item may appear in a tuple more than once, its frequency is recorded and used later on.

4.2. Assessing block quality

In Section 3.1 we presented σ^i, a generic score function over tuples in a block. We now present a concrete instantiation that is used in the algorithm. Intuitively, we would like to give a higher score to blocks with many common items, and a few uncommon items. Uncommon items may be the result of inaccurate and missing values. Also, tuples in a block may share items of limited discriminative power, *e.g.*, a certain item may appear in half of the database tuples. Finally, the differences between tuples may surpass the commonality represented by the block.

To assess the quality of a block we use a version of the extended Jaccard similarity [16], which accounts for similar (as well as exact) q-grams. Let B^i be a block with support P^i. $M^i = \uplus_{t \in P^i} t.items$ is the itembag of all tuples in P^i, where $\uplus$ stands for the bag union operation. Recall that $t.items$ represents the q-grams of a tuple t.

We now define the aggregated similarity of an item $j \in M^i$ to its support set P^i, denoted $\alpha_i(j)$. Intuitively, $\alpha_i(j)$ should reflect the strength of relationship of item j with the block B^i. Items with a high aggregated similarity metric indicate a larger commonality among tuples in a block.

Definition 4. *The aggregated similarity of an item j in a block B^i is:*

$$\alpha_i(j) = \frac{1}{|P^i|} \sum_{t \in P^i} \max_{k \in t.items} Sim(k, j)$$

where $Sim(k, j)$ is a pairwise string similarity metric.

We use the Jaro-Winkler method [18] in our experiments to compute $Sim\,(k, j)$. With the proposed method, items in E^i receive an aggregated similarity of 1.0 because such items appear in each and every tuple in P^i. However, we also give a relatively high score to items that have similar items in a large part of P^i. Therefore, the itemset M^i is divided into two disjoint groups. First, the set of *common items*, denoted C^i, is the itemset whose aggregated similarity metric (Definition 4) is above some predefined

threshold Sim_th. For $Sim\_th = 1$, $C^i = E^i$ since each item in E^i is common to all tuples in P^i and was thus chosen by the MFI algorithm. The remaining items P^i/C^i are *uncommon items*, items whose aggregated similarity metric is below Sim_th.

Example 4. *Consider the block B^i with $P^i = \{t_1, t_2\}$ from Table 2 and assume the use of 3-grams and $Sim\_th = 0.9$. All items except 'address1' and 'suburb' are identical, and will be classified as common items. The 3-gram 'mur', originating from t_1's 'suburb' attribute will be classified as a common item because it is relatively close to the 3-gram 'mud' originating from t_2's 'suburb' attribute (Sim('mud','mur') ≥ 0.9). Therefore, the aggregate similarity metric for 'mur' passes the aggregate similarity threshold and is considered a common item.*

We propose to set a score using the tf/idf weighting scheme [19], a common measure in Information Retrieval that measures both commonality and its significance. The tf/idf score of an item j in block B^i is computed as follows:

$$v_i(j) = log(tf_i(j) + 1) \cdot log(idf_i(j)), \tag{1}$$

where $tf_i(j)$ is the total number of times an item j appears in the bag M^i (if item $j \in E^i$ then $tf_i(j) \geq |P^i|$). $idf_i(j) = \frac{|D|}{n_j}$ where n_j is the number of tuples in database D that contain item j. The tf/idf weight for an item j is high if j appears many times in the support set P^i (large $tf_i(j)$) and j is a sufficiently rare item in the database (large $idf_i(j)$). The above score is calculated for all distinct items in M^i.

The lexicon, discussed in Section 4.1, may be used to add metadata to the ids which represent the items. This metadata may include relative weights indicating the quality and discriminative power of the attributes from which the items originated. We represent the weight of item $j \in M^i$ as $w(j)$. These weights offer an opportunity to inject expert knowledge into the process. An expert, for example, can identify some attributes to be more important than others, which is reflected in assigning higher weights to items of these attributes. In the absence of expert knowledge (the setting we have assumed in our empirical study) weights are set to be equal. Combining the tf/idf score and item weights in the extended Jaccard similarity function [16] results in the following score function that measures the similarity among members of a cluster rather than the similarity between two strings in the Jaccard score function [7]:

$$\sigma^i(P^i) = \frac{\sum_{j \in C^i} w(j) v_i(j) \alpha_i(j)}{\sum_{i \in M^i} w(j) v_i(j)} \tag{2}$$

4.3. Selection of minsup

The $minsup$ parameter sets the minimum number of tuples in a block ($|P^i|$). If $minsup$ is set too high then no MFIs may be mined, for the lack of commonality among members of clusters of large size. Another scenario is that large clusters may share some commonality, which may be small or insignificant, resulting in a low quality clustering that masks the true match among cluster members. On the other hand, if $minsup$ is set too low, then larger clusters may be missed, resulting in low recall. This happens whenever inside a larger cluster there may be groups of tuples with higher

commonality, causing the generation of larger MFIs with a smaller support size (*i.e.*, less tuples in each cluster).

The following two observations guide our treatment of the $minsup$ parameter. First, we observe that duplicate tuple sets may vary in size. For example, in the Cora dataset (see Section 5) there is a big variation in the number of duplicate tuple sets, representing the same real-world entities, with some duplication sets having more than fifty tuples. Our second observation is that in many scenarios some information on the size of duplicate sets may be known in advance. For example, when integrating two clean sources, each tuple may have at most one duplicate and $minsup$ can be set to 2.

Based on these two observations, the algorithm is designed to proceed in an iterative fashion. The initial minimal support is set to be an upper bound on the expected duplication set size. At each iteration, after the high-scoring clusters are discovered and processed, the tuples belonging to high-scoring clusters are removed from the dataset, and the algorithm continues with smaller thresholds on smaller datasets. Such a design also enables an improved performance since smaller support thresholds yield more MFIs and therefore starting by mining MFIs with the largest expected support size results in smaller datasets for lower thresholds.

Example 5. *As an example, consider Table 2 once more. There are duplicate sets of sizes 6, 4, and 2. Executing an MFI algorithm with* $minsup = 6$ *returns only the large duplicate size and not the MFIs corresponding to the size-4 and size-2 clusters, due to lack of support. After the big cluster is discovered, we can remove the detected tuples and rerun the MFI mining algorithm on the remaining tuples with a lower* $minsup = 4$ *and discover duplicate clusters of size 4 (tuples 9-12 in our running example). Iteratively, the algorithm can be rerun on the leftover tuples with* $minsup = 2$ *to discover blocks with support of* 2 *tuples.*

4.4. Applying the SN and CS criteria

Following the desirable properties of the CS and SN criteria [6] introduced in Section 3, we now show the use of these criteria in the algorithm configuration. The obvious way to decide on block acceptance is to define a score acceptance threshold, min_th, which turns out to have a dramatic effect on the quality of chosen blocks. A threshold set too low leads to low precision while a threshold set too high results in a decrease in recall. The main challenge with setting min_th is the high dependency on the error characteristics of the data, a feature that may be hard to recognize and dramatically changes among datasets. On the other hand, the configuration of the SN criterion, p, is simpler as it represents a generic approach towards the blocking process that is translated to a limit on a tuple's neighborhood. For example, setting the parameter p to 1.5 when $minsup = 2$ limits the number of pairwise comparisons a tuple will undergo to 3. Our empirical analysis indicates that in most cases a value $p \in [1.5, 4]$ yields satisfactory results.

As it turns out, the two parameters, min_th and p, are tightly connected. To be more precise, configuring the sparse neighborhood constant, p, induces a minimum threshold min_th value. The reason for this is as follows: p constraints the neighborhood of each tuple in the output blocking, resulting in the elimination of low scoring blocks (according to Eq. 2). Therefore, the algorithm seeks the smallest threshold,

min_th, such that the set of blocks that satisfy $S^i \geq min\_th$ also satisfy the constraint $max_{t \in D}|N(t)| < p \cdot minsup$.

This dependency enables setting the score threshold by default to 0. With the appropriate p, min_th automatically rises to a level that complies with the SN criterion. An administrator may wish to constrain the scores of the selected blocks in addition to p, *e.g.*, some tasks may require only high scoring blocks. In such a case, the min_th parameter may be configured manually to a strictly positive value. We consider this scenario to be the exception rather than the rule.

4.5. The algorithm pseudocode

The pseudo-code of MFIBlocks, an MFI-based blocking algorithm, is detailed in Algorithm 1. The algorithm receives as input the tuple database, D, an initial support $minsup$ set to the largest expected duplicate cluster in D and its decrement step from one iteration to the next st. The algorithm is also provided with the sparse neighborhood constant, p, and an optional absolute minimum score threshold min_th. min_th can mostly receive a default value of 0 because the algorithm automatically prunes blocks of low score due to the violation of the sparse neighborhood constraint p.

The algorithm runs in iterations, where in each iteration (lines 2-31) the MFI mining algorithm is run with a smaller minimum support. Each such MFI run creates a set of blocks. Tuples belonging to a block that complies with the sparse neighborhood and minimum threshold criteria, are considered $minsup\text{-}covered$. They are marked for further processing and are then removed from the database. The algorithm continues until either all the tuples in D are covered or $minsup$ falls below 2. In each additional iteration, the MFI mining algorithm is run on the set of uncovered tuples, returning a new set of tuple sets (line 4).

Following the MFI mining step, blocks are extracted and their size is checked. Blocks with support size larger than $p \cdot minsup$ do not comply with the sparse neighborhood constraint, and are therefore discarded (lines 10-12). Otherwise, the block's score is calculated using the cluster Jaccard similarity (see Eq. 2). The block is kept if its score passes the min_th threshold (lines 13-16).

At this point we illustrate how the compact set and sparse neighborhood criteria, discussed in Section 4.4, are implemented in the algorithm. We are interested in finding a set of blocks of maximum cardinality, where each block maintains a score above min_th and that the sparse neighborhood constraint is satisfied ($max_{t \in D}|N(t)| \leq p \cdot minsup$). In order to achieve this we keep track of the candidate pairs and adjust the min_th dynamically. Once the algorithm encounters a block B^i with score $S^i > min\_th$, the distinct tuple pairs in the block are recorded (line 18). If the sparse neighborhood criterion is violated for some tuple $t \in P^i$ we remove from consideration all candidate pairs originating from blocks whose score is below the lowest scoring block to which tuple t belongs. This means that the increase of the min_th is based on the value of $\min_{B^i|t \in P^i} S^i$. Finally, the set of returned blocks corresponds to the lowest score that complies with the sparse neighborhood criterion. It is worth noting that the neighborhood growth constraint may be left unsatisfied for a certain $minsup$. In such a case, the search for threshold min_th will be set to 1.0, alerting that no blocking exists that satisfies the constraint.

Algorithm 1: MFI based blocking

Input: Database D; initial support size $minsup$; step $st > 0$ for decreasing $minsup$; NG limit p;
Optional Input: minimum threshold min_th (default is 0)
Output: (Sparse) matrix $M[i, j]$ containing the tuple pairs to be compared by an Entity Resolution algorithm

1: $P = \emptyset$ {currently covered tuples in D}
{repeat until all tuples are covered or $minsup$ falls below 2}
2: **while** $(P \neq D \; AND \; minsup \geq 2)$ **do**
3: $candidateBlocks \leftarrow \emptyset$
4: $MFIs \leftarrow MFI(D \backslash P, minsup)$ {run the MFI algorithm on the uncovered tuples}
5: **for all** $E^i \in MFIs$ **do**
6: $P^i \leftarrow \{r \in D : E^i \subseteq r.items\}$ {calculate the support of E^i}
7: $candidateBlocks \leftarrow candidateBlocks \cup P^i$
8: **end for**
9: **for all** $cB \in candidateBlocks$ **do**
10: **if** $size(cB) > p \cdot minsup$ **then**
11: ignore cB and continue
12: **end if**
13: $score \leftarrow ClusterJaccard(cB)$
14: **if** $(score < min\_th)$ **then**
15: ignore cB and continue
16: **end if**
17: **for all** $(r_i, r_j) \in cB$ **do**
18: mark (r_i, r_j) as a candidate pair
19: **if** $(NG(r_i) > p \cdot minsup)$ **then**
20: $min\_th \leftarrow MAX(min\_th, min_{B^k|_{i \in P^k}} S^k)$
21: **end if**
22: **if** $(NG(r_j) > p \cdot minsup)$ **then**
23: $min\_th \leftarrow MAX(min\_th, min_{B^k|_{j \in P^k}} S^k)$
24: **end if**
25: **end for**
26: fetch marked candidate pairs belonging to clusters with $score > min\_th$
27: update $M[i, j]$ with the marked pairs
28: **end for**
29: $P \leftarrow P \cup$ newly covered tuples from $M[i, j]$
30: $minsup \leftarrow max(minsup - st, 0)$
31: **end while**
32: **return** $M[i, j]$

At the end of each iteration, the tuples that belong to a block in the finalized block set are added to the set of covered tuples (line 29).

As a final remark, we note that Algorithm 1 may be easily adjusted to run on any encoding of the database, including phonetic schemes such as Soundex. This type of encoding can replace the encoding of the attribute values as q-gram ids.

4.6. Improving Performance

The bottleneck of the proposed algorithm is the processing of a large amount of MFIs (lines 4-20 in Algorithm 1). The problem of enumerating the set of maximal frequent itemsets is #P-complete [11] (see discussion in Section 3), which is also the worst case running time of Algorithm 1. We observe that the number of MFIs depends on the number and frequency of items in the dataset. A rare item that belongs to a small

number of tuples can participate in only a few MFIs. For example, an item contained in two tuples may participate in at most a single MFI when $minsup = 2$. Using the same support threshold, an item contained in 5 tuples may participate in at most 10 MFIs. Highly frequent items, therefore, may combine to create a larger number of MFIs. In our setting, such items provide little information for two reasons. First, MFIs containing highly frequent items have a larger chance of being supported by a large number of tuples. In many cases such MFIs violate the neighborhood growth constraint and are discarded even before their score is calculated (lines 10-11 in Algorithm 1). Even if such an MFI satisfies the constraint, due to the high frequency of its members, it would likely score low and be discarded (lines 14-15) after having used precious resources to compute its score. Highly frequent items also receive a low tf/idf score, and thus have little impact on the overall score even for MFIs that pass the threshold.

Given the observation above, we propose a method for pruning a small percentage of the most frequent items as a preprocessing step. During the initial data processing, we count the number of occurrences of each q-gram. We can then remove a certain percentage of q-grams (in our experiments, see Section 5.2.3, we found that pruning up to 0.3% of q-grams is sufficient to yield good results) before running the algorithm.

5. Experiments

This section presents an extensive empirical comparison of Algorithm 1 with other blocking algorithms. We describe the experiment setup in Section 5.1. Section 5.2 details and discusses the experiment result using both real-world and synthetic datasets with several error characteristics. The dataset sizes are in the range of 1000 to 2 million tuples. The experiments demonstrate both the effectiveness of the proposed approach with regard to known quality metrics (precision, recall and F-measure), and its practicality in terms of runtime. In particular, we show that the effort that is invested in the blocking stage pays off when it comes to the comparison stage.

The algorithm was implemented in Java using JDK 1.6, and the experiments were run on a Linux server with four 3.30GHz Intel Core processors with 32GB RAM.[1]

5.1. Experiments setup

5.1.1. Datasets

We ran the proposed algorithm on both real-world and synthetic datasets. The real-world datasets include the benchmarks Restaurants, Census, and Cora datasets from the SecondString toolkit.[2] We also experimented with the CDDB dataset,[3] a randomly selected sample of CDs from freeDB.[4] Each CD entry contains the name, artist, category, genre, and year fields along with the track titles. Following Draisbach and Naumann [20], we made use of all available fields, but disregarded all but the first

[1] The binaries and execution instructions are available for download at http://www.technion.ac.il/~batyak/MFIBlocksImp.zip

[2] Available from: http://secondstring.sourceforge.net

[3] http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_Naumann/projekte/dude/cd.csv

[4] http://www.freedb.org

DB Name	size	# of matching pairs	Domain	# of attributes	prefix	max # of duplicates per tuple
Restaurants	864	112	Restaurant guide	5	none	1
Cora	1295	17184	Bibliographic	12	30	81
Census	842	344	people	5	none	1
CDDB	9763	299	music CDs	6	none	5
Clean	1K-2000K	200-40K	names and addresses	12	5	3
Dirty	1K-2000K	400-430K	names and addresses	12	5	9

Table 4: Datasets

track. Draisbach and Naumann set the blocking key to the first three letters of the artist, title, and first track. In our approach, the key was selected automatically. Table 4 summarizes the dataset features.

We generated synthetic datasets of sizes 1K, 5K, 10K, 50K, 100K, 200K, 500K, 1000K, and 2000K tuples using the FEBRL [15] data set generator. For comparison, we generate the datasets according to parameters specified by Christen [3]. Tuples are first created based on frequency tables containing real-world names (given and surname) and addresses (street number, name and type, postcode, suburb, and state names). A random generation of duplicates of these tuples is based on modifications (*e.g.*, inserting, deleting or substituting characters, and swapping, removing, inserting, splitting or merging words), and real-world error characteristics. We generated two datasets for each size, with different error characteristics, as follows. *Clean datasets* contain 20% duplicates with up to three duplicates per tuple, one modification per attribute at most, and up to three modifications per tuple. *Dirty datasets* contain 40% duplicate tuples with up to nine duplicates per tuple, no more than three modifications per attribute, and up to ten modifications per tuple.

5.1.2. Parameter settings and evaluation metrics

In our experiments, we have varied three parameters of Algorithm 1 to test their impact on effectiveness and efficiency: 1) the SN parameter p; 2) the initial support size $minsup$; and 3) the *optional* parameter, minimum score threshold min_th.

For each dataset we ran two versions of the experiment. The first, without an SN constraint, is effectively achieved by setting p to be large ($p > 100$). Therefore, all blocks that pass the minimum threshold min_th are included in the final blocking. In the second version, we set $p \in [1.5, 4]$, which was shown in our preliminary experiments (not shown here for lack of space) to be a range that yields good results.

In databases with long attribute values, we applied Algorithm 1 on value prefixes. This highly increased performance with little effect on the evaluation metrics. The same prefix was applied to *all* attributes in a dataset.

When comparing our algorithm to 12 other blocking techniques implemented in FEBRL [15], we have relied on the authors' selection of blocking keys as well as other parameters (similarity function, similarity threshold, q-gram size) that were set in [3]. Therefore, we compared our results to the results achieved by using three different blocking keys that were defined per dataset using a variety of combinations of tuple attributes using domain experts. For Algorithm 1 we included *all* attributes in the dataset with *equal weights*, eliminating the need to manually design a blocking key.

The blocking algorithms were assessed using three measures, namely Recall (known in the entity resolution literature as Pairs Completeness PC), Reduction Ratio (RR) [21] and Precision (known as Pairs Quality PQ) [3]. Recall (PC) measures the coverage of true positives, *i.e.*, the number of true matches in blocks versus those in the entire database. Let P_B be the number of matching pairs located in blocks returned by a blocking algorithm and let P_D be the total number of matching pairs in the database. Then recall is calculated as $PC = \frac{P_B}{P_D}$. Precision (PQ) is the number of true positives divided by the total number of generated tuple pairs. A high PQ value means that the blocking algorithm is effective and mainly generates truly matched tuple pairs. On the other hand, a low PQ value indicates a large number of generated non-matches, resulting in more tuple pair comparisons, which is computationally expensive. Let C be the number of comparisons in blocks generated by a given blocking algorithm. Then precision is calculated as $PQ = \frac{P_B}{C}$. The F-measure is the harmonic mean of recall (PC) and precision (PQ): $F = \frac{2PC \cdot PQ}{PC+PQ}$.

The Reduction Ratio measure quantifies how well a blocking algorithm prunes candidate pairs to be compared. Let N be the number of possible comparisons between tuples in a dataset. Then, $RR = 1 - \frac{C}{N}$. In the absolute majority of our experiments, over all blocking algorithms, $RR > 0.9$, sometimes reaching 0.99. Therefore, it does not serve as a discriminative measure and we focus in our analysis on precision (PQ) and recall (PC), and its derivative, the F-measure (F).

5.2. Experiment results and analysis

We now present and analyze the results of our experiments. We start with the four real-world datasets, analyzing each of them separately (Section 5.2.1). We then move to synthetic data to assess the impact of various control parameters on performance (sections 5.2.2 and 5.2.3). Finally, Section 5.2.4 compares the combined blocking and comparison stages runtime results with existing works in the literature.

We start with a few observations on the performance of Algorithm 1. The sparse neighborhood constraint p has little effect on recall (PC), which remains high for almost all tested p values. On the other hand, precision (PQ) was drastically improved when limiting p to be in the range of $[1.5, 4]$. We observe that p compensates for a low threshold parameter (min_th). This behavior exhibits the usefulness of the proposed approach. It demonstrates that the knowledge regarding the expected size of duplicate clusters, which is easier to asses, can replace the harder task of configuring similarity thresholds. Since the behavior described above applies to almost all of the datasets, we show only results for $p \in [1.5, 4]$.

5.2.1. Real-world datasets

We now compare the performance of Algorithm 1 with the blocking algorithms surveyed in [3] on three real-world datasets. Each blocking algorithm was run with three different blocking keys and various parameter settings. For each dataset, we compare Algorithm 1 to the *highest* performing algorithm on that dataset, in terms of its F-measure, as reported by Christen. For the Restaurants dataset it is the Suffix Array blocking (SuAr) algorithm [22], for the census dataset it is the threshold-based canopy clustering (CaTh) algorithm [2] (closely followed by the standard blocking

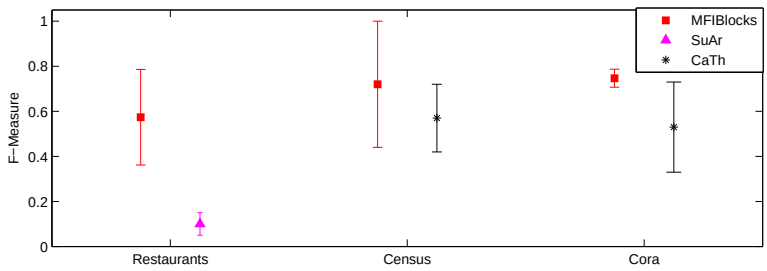


Figure 2: Real-world datasets: average and standard deviation

algorithm), and for Cora it is also the threshold based canopy clustering algorithm. All other algorithms in the survey demonstrated worse performance.

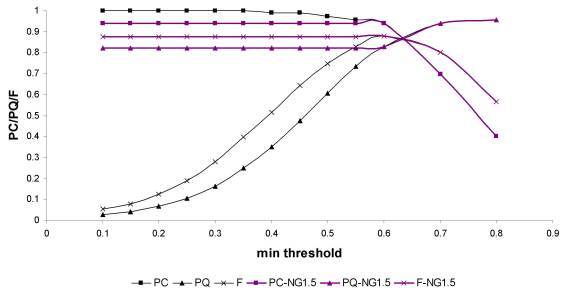
Figure 2 provides a comparison of the average F-measure and standard deviation of the best performing algorithms [3] and Algorithm 1. The average for Algorithm 1 was taken over all NG constraints and all minimum score thresholds. Algorithm 1 is better, on average, where in both the Restaurant and Cora datasets this improvement is statistically significant. In particular, the improvement is apparent in the Restaurants dataset, where the F-measure of Algorithm 1 was six times better, on average, then the best performing algorithm in the survey.

Figures 3a-3c provide a graphic illustration of the effectiveness of Algorithm 1 when run under various similarity thresholds (min_th) for the three real-world datasets. The precision (PQ) measure in Figure 3a starts low for low thresholds and increases significantly with higher thresholds. The addition of the sparse neighborhood constraint (marked as PQ-$NG1.5$), which bounds the neighborhood growth of the tuples, increases precision dramatically for low thresholds. We observe the same phenomenon in all of our experiments. Therefore, we present from now on only results where the sparse neighborhood constraint is in effect.

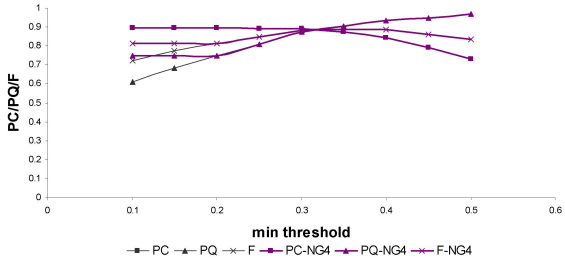
We note that for datasets with large clusters (*e.g.*, in the Cora dataset some duplicate sets are of size greater than 50), blocks are created using a large minsup and can satisfy the NG constraint even for low thresholds. Therefore, the effect of the NG constraint in such cases is moderate, as demonstrated in Figure 3c.

A trade-off of precision (PQ) and recall (PC), indicated as a peak in the F-measure, is achieved for different thresholds in different datasets. It is 0.6 for the Restaurants dataset (Figure 3a), 0.3 for the Census dataset (Figure 3b), and 0.2 for Cora (Figure 3c). We conclude that different datasets require different thresholds in order to achieve quality blocking. This threshold mainly depends on the data error characteristics, which may be difficult to foresee. Therefore, instead, we configure the neighborhood growth parameter p.

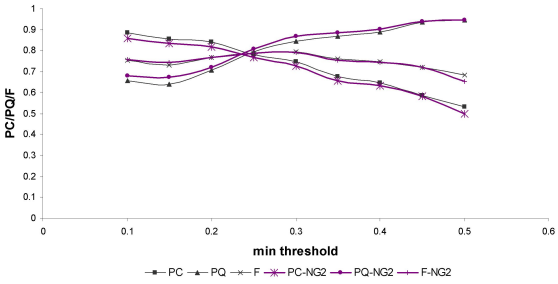
We also ran Algorithm 1 on the CDDB dataset. In Figure 4, we plot the required number of comparisons needed to discover a particular number of duplicates using varying min_th values in the range 0.0-0.8. Each min_th value induces a certain number of comparisons and identifies a certain number of duplicates. Generally speaking, higher min_th values lead to a smaller number of blocks and thus to a lower number of comparisons. It is worth noting that the graph does not show a monotonic behavior.



(a) Restaurants Dataset



(b) Census Dataset



(c) Cora Dataset

Figure 3: Real-world datasets

For example, 19,120 comparisons lead to the discovery of 270 duplicates while 37,245 lead to the discovery of 250 duplicates only. The reason for this nonmonotonic behavior has to do with the initial setting of the $minsup$ value. When a low threshold is used, a false positive cluster may be created at an earlier iteration. As a result, the members of this cluster are removed from the dataset and are not processed further (line 29 in Algorithm 1), ultimately reducing the number of discovered duplicates.

We now compare the performance of Algorithm 1 with the results reported by Draisbach and Naumann [20], using the sorted blocks algorithm. This algorithm is based on the sorted neighborhood algorithm, [23, 1] whose performance is recorded as inferior to other algorithms [3]. All-in-all, this dataset has 299 duplicates. To reach the first 254 duplicates (85% of the duplicates), Algorithm 1 needed 1,945 comparisons and to reach 270 duplicates (90% of the duplicates), Algorithm 1 needed 19,120

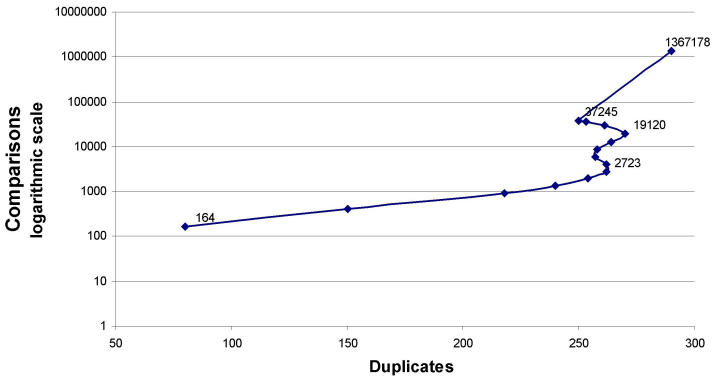


Figure 4: The CDDB dataset

comparisons. The sorted blocks algorithm required about 5 times more. The sorted blocks algorithm managed to reach at most 277 duplicates (93%) using more than a million comparisons while Algorithm 1 reached 290 duplicates (97%). Here, the cost was high (1,367,178 comparisons). We note that for both algorithms there is an exponential growth of effort in attempting to reach the final set of duplicates in this dataset. Algorithm 1, however, managed these resources more effectively.

5.2.2. Synthetic datasets

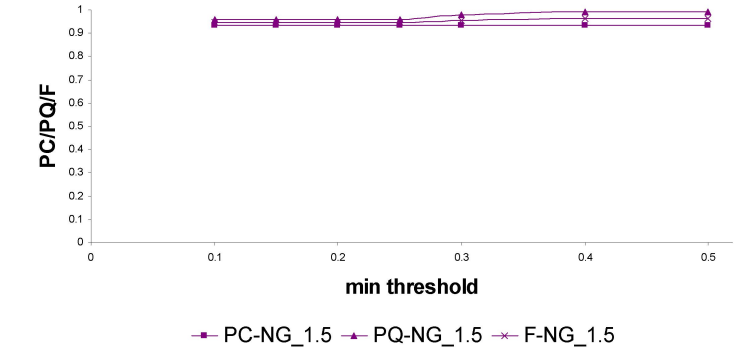
We ran our algorithm on the synthetic datasets described in Section 5.1.1 with $prefix = 5$. We experimented with similarity threshold values in the range of $[0.1, 0.5]$ for both clean and dirty datasets. We present results for datasets of $1,000$ and $100,000$ in figures 5 and 6, respectively.

In the clean datasets (Figure 5), precision, recall and F-measure remain almost constant and above 0.92 for all min_th values when using a neighborhood growth parameter of $p = 1.5$.

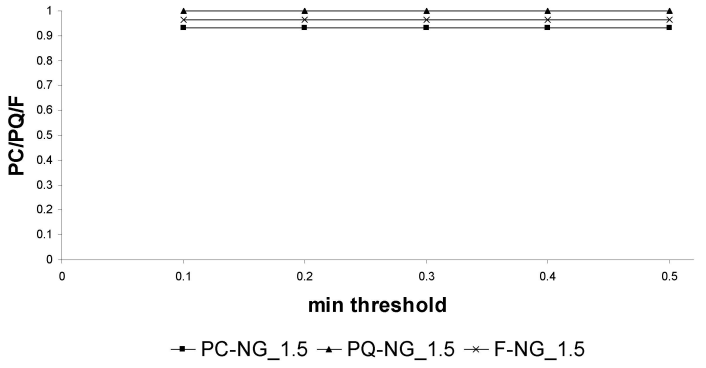
The clean and dirty datasets exhibit a similar pattern except for recall that begins to decline for the latter for thresholds higher than 0.4. This is an expected phenomena because in dirty datasets duplicate tuples are less similar to one another.

Figure 7 compares the F-measure of Algorithm 1 with other blocking algorithms [3] on synthetic datasets. For each dataset, we compare Algorithm 1 to the *highest* performing algorithm on that dataset, in terms of its F-measure, as reported by Christen. For the datasets of sizes $1K$ and $10K$ it is the $q\text{-}gram$ based indexing approach (QGr) [24]. For the clean $100K$ dataset it is the Blocking approach (Blo) [25]. For the dirty $100K$ dataset it is the Threshold Based Canopy Clustering algorithm (CaTh) [2].

Experiments analysis reported by Christen [3] indicates that the effectiveness of all blocking approaches begins to decline with the size of the dataset (see Figure 7). There, algorithms that employ fixed limits on the block sizes (*e.g.*, fixed window size for the



(a) Clean 1000

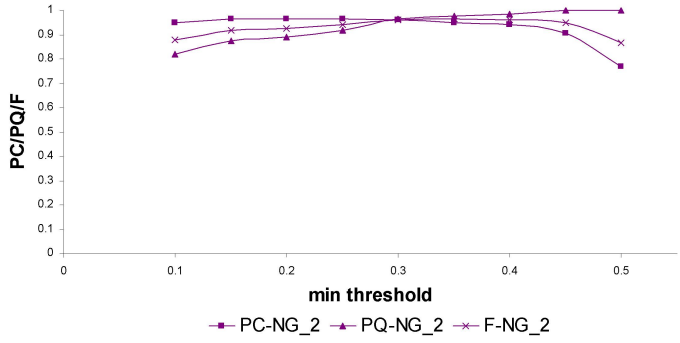


(b) Clean 100000

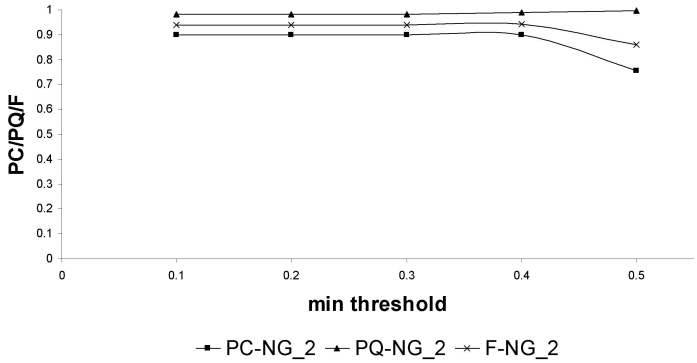
Figure 5: Performance: clean datasets

sorted neighborhood approach and fixed number of neighbors in the canopy clustering approach) will experience a loss in recall as the dataset becomes larger. The reason for this is that despite the fact that a larger number of tuples share the same blocking key, due to the block size limitations they will be placed in different blocks leading to loss in recall. With approaches that do not place such a limit, the increasing block sizes will bring to a rise in the number of candidate tuples pairs, lowering precision.

Figure 7 shows that Algorithm 1 is more robust, maintaining a high level of F-measure regardless of the dataset size. Such robustness can be attributed to the lack of an upper limit on block size and therefore avoiding the decline in recall. In addition, the MFI approach "automatically" adjusts the blocking key such that the created blocks are small enough even when the dataset is large. If the dataset is large enough then the minimum support may be achieved by tuples exhibiting a higher similarity than possible with a smaller dataset. Therefore, for larger datasets Algorithm 1 yields a larger number of appropriately sized blocks (depending on the $minsup$ parameter) but overall the precision and recall does not change much.



(a) Dirty 1000



(b) Dirty 100000

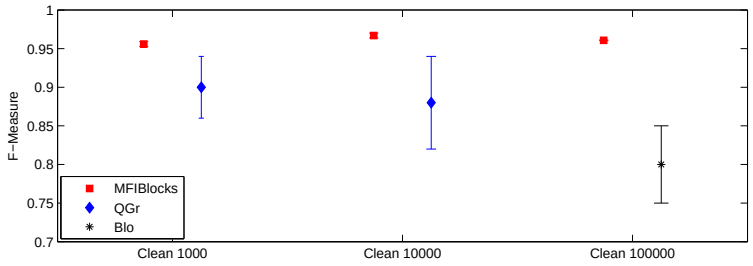
Figure 6: Performance: dirty datasets

5.2.3. Scalability and runtime analysis

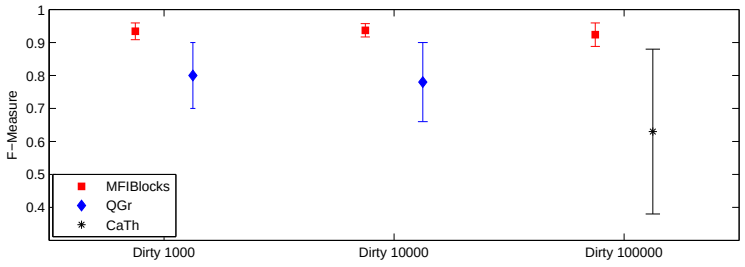
We experimented with different ways of reducing the number of MFIs, as discussed in Section 4.6, and tested the algorithm for effectiveness and performance. The following set of experiments was run on dirty synthetic datasets with sizes in the range of $10K$-$2000K$, applying attribute prefix (see Section 5.1.2) and using 4-gram tokenization (see Section 4.6).

Pruning frequent items allows a trade-off between efficiency and effectiveness (see Section 4.6 for the detailed discussion). Figure 8 summarizes the results in terms of F-measure (Figure 8a) and in terms of runtime (Figure 8b). The x-axis in both diagrams represent the percentage of pruned items. The y-axis in Figure 8a shows the F-measure achieved with datasets of varying sizes and the percentage of pruned items. The y-axis in Figure 8b shows, on a logarithmic scale, the runtime, measured in seconds.

While improved runtime results in a reduced F-measure, we note that even a small percentage of pruning, resulting in only a moderate decrease in F-measure, yields a significant improvement in runtime. For example, the runtime for the algorithm on the



(a) Clean datasets



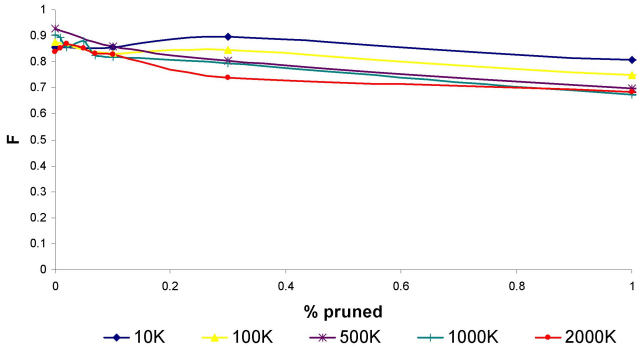
(b) Dirty datasets

Figure 7: Synthetic datasets: average and standard deviation of F-measure

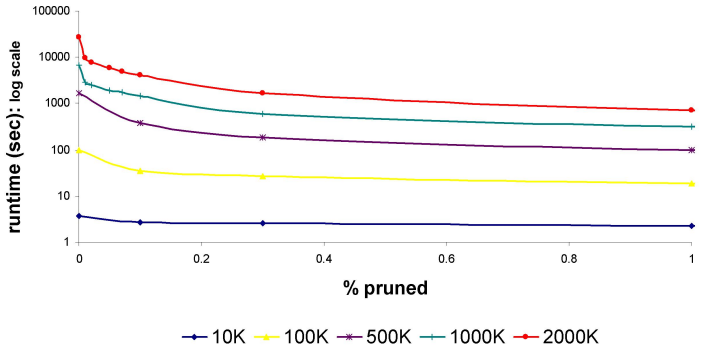
$1000K$ dataset drops from 6860 to 2860 seconds (reduction of 58%) by pruning only 0.01% of the most frequent items, at the cost of a slight drop of F-measure, from 0.905 to 0.893. It becomes evident from both diagrams that pruning up to 0.3% of the more frequent items captures the majority of saving in runtime while maintaining a high F-measure. Beyond 0.3%, no significant improvement of runtime is achieved and the reduction in F-Measure becomes much more prominent.

To demonstrate the significant improvement in runtime, Figure 9a displays the runtime as a function of the database size for three different pruning thresholds. The 0% curve represents no pruning, while the other two curves represent pruning of 0.1% and 0.3%. As the dataset grows, the difference between the top curve and the lower two curves becomes more prominent, demonstrating the practicality of pruning. Runtime grows quadratically as a function of the database size for all levels of pruning (with R^2 values for a quadratic trendline greater than 0.99) but with a very low coefficient for the quadratic component ($< 10^{-8}$). It is worth noting that the differences between the coefficients of the three settings are extremely small, yet their impact is evident. Figure 9(b) shows the F-measure corresponding to pruning thresholds 0-0.3%. As can be seen, the F-measure remains above 0.75 for all pruning thresholds, with a rather small decrease in F-measure with 0.1% and 0.3% pruning levels.

Figure 10 illustrates the tradeoff between runtime and F-measure for datasets of size $500K$,$1000K$, and $2000K$. Each point in the graph represents an experiment with a different pruning level. For all three datasets, there is a range of less than 2000 sec-



(a) % pruned vs. F-measure



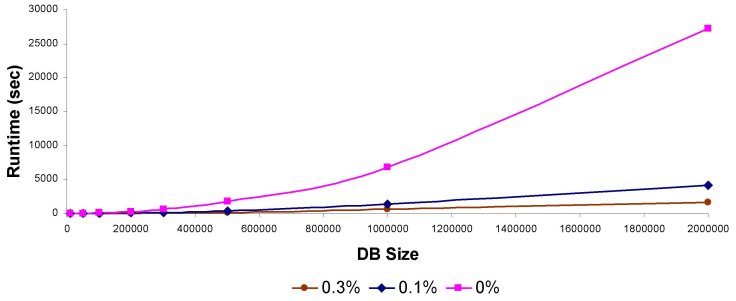
(b) % pruned vs. runtime (logarithmic scale)

Figure 8: Effectiveness and performance for the dirty synthetic datasets

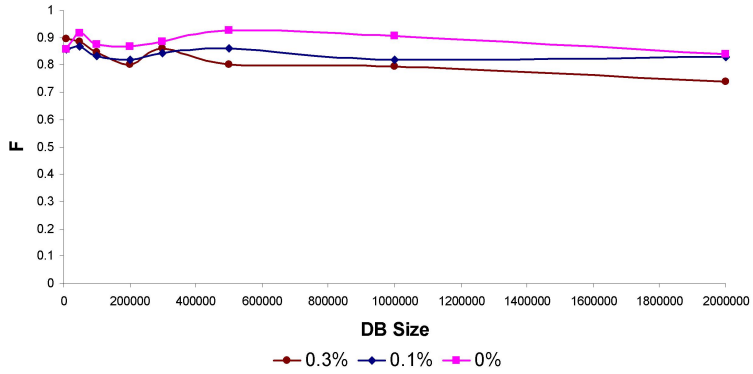
onds where additional investment of time yields significantly higher F-measure levels. After that, less pruning (and therefore more time invested) does not make a significant difference. Therefore, we can conclude that low pruning levels ($\leq 0.1\%$) serve as a good tradeoff level.

5.2.4. Combined runtime analysis

The overall runtime of an entity resolution process depends largely on the number of tuple pairs generated by the blocking stage and the comparisons performed in the comparison stage. Therefore, investing more time in the blocking stage may lead to the generation of higher quality blocks, with a lower number of candidate pairs, possibly making the overall process more efficient. In this section we provide a comparison of the combined blocking and comparison runtime performance. The main conclusion we can draw here is that Algorithm 1 is mostly comparable with the runtime results reported in the literature while providing an easy-to-use method that yields high quality blocks, which in turn can translate to high quality resolved databases.



(a) % pruned vs. runtime



(b) % pruned vs. F-measure

Figure 9: Effectiveness and performance for low pruning thresholds

We applied five different blocking techniques on two datasets. The Restaurants dataset as a representative of a real-world dataset and a $10K$ synthetic dirty dataset, which was used in our experiments before (see Section 5.2.2). All experiments were conducted using FEBRL [15] and followed the guidelines provided by Christen [3]. In particular, we have chosen a rather small synthetic dataset to allow most of the blocking algorithms in FEBRL to run on it.

Algorithm 1 does not require a-priori key specifications. For the other blocking algorithms in the experiment we used, for the Restaurants dataset a single blocking key combined of the first four characters of the city attribute concatenated with the first four characters of the type attribute. In the comparison stage we used the name, address, city and type fields using standard string comparison functions edit distance, Jaro and Winkler. In the classification stage we used the Fellengi Sunter [25] approach. As expected, the majority of the runtime was taken by the comparison stage.

For the synthetic dataset we defined three different settings, marked as $BK1$, $BK2$, and $BK3$ in Table 6. In the first setting, denoted $BK1$, we created a single index whose

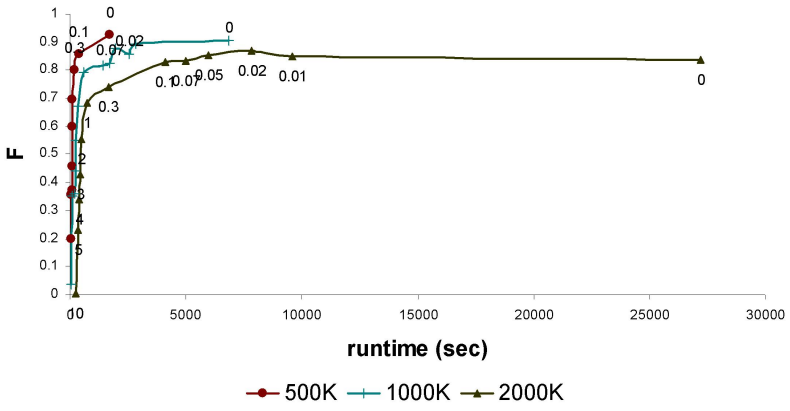


Figure 10: F-measure vs. runtime (sec)

blocking key is the surname attribute encoded in double metaphone. In the second setting, denoted $BK2$, we created a single index whose blocking key is the first three characters from the soc_sec_id attribute concatenated to the first three characters of the phone attribute. In the third setting, denoted $BK3$, we created two indexes, the first corresponds to the surname attribute, and the second to the suburb attribute concatenated with the postcode attribute. All values were encoded in double-metaphone.

In the comparison stage we used the given_name, surname, address_1, address_2, suburb, state and soc_sec_id attributes using standard string comparison functions [26]. In the classification stage we used the Fellengi Sunter [25] approach. As expected, the majority of the runtime was taken by the comparison stage. Table 6 presents the overall runtimes of the various blocking techniques in these settings.

Blocking technique	**Parameters**	**# candidate pairs**	**runtime (sec)**
Blocking		7259	6.73
Suffix Array [22]	min length = 3, max block size = 6, Suffix Only	481	0.891
Canopy Index [2]	TF-IDF, Global thresholds: 0.8,0.9 3-grams	7259	7.48
Sorting Index [1]	window=5	24,196	23.44
Sorting Index [1]	window=3	16,765	15.61
String-Map Index [27]	Global thresholds: 0.8,0.9, Grid Size = 100, mapping dimension=15, Similarity function=largest common subsequence	7,234	8.08
Algorithm 1	$p = 2$, $minsup = 2$, $min\_th = 0.1$	246	4.23

Table 5: Blocking and comparison stages on the Restaurants dataset

The parameter setting and runtime results of the blocking and comparison stages for the Restaurants dataset appear in Table 5. The table presents the number of candidate pairs created by the respective blocking algorithms and the total combined runtime of the blocking and comparison stages. The results show that the time invested in blocking by Algorithm 1 has paid off by yielding a smaller set of candidate pairs. Algorithm 1 is ranked here first in terms of number of candidate pairs and second only to Suffix Array in terms of combined runtime. It is also worth noting that Figure 2 shows a statistically significant improvement in F-Measure of Algorithm 1 over Suffix Array.

Blocking technique	**Parameters**	**# candidate pairs BK1 / BK2 / BK3**	**runtime (sec) BK1 / BK2 / BK3**
Blocking		$157,579$ / $15,848$ / $240,453$	55 / 6.44 / 86
Suffix Array [22]	min length = 3, max block size = 10, Suffix Only	$9,763$ / $17,303$ / $18,833$	4.03 / 6.34 / 7.27
Canopy Index [2]	TF-IDF, Global thresholds: 0.8,0.9, 3-grams	$157,583$ / $40,857$ / $240,459$	52 / 16.49 / 87
Sorting Index [1]	window=5	$271,479$ / $94,487$ / $509,594$	102 / 37 / 205
Sorting Index [1]	window=3	$212,205$ / $58,078$ / $376,990$	78 / 23.11 / 151
String-Map Index [27]	Global thresholds: 0.8,0.9, Grid Size = 100, mapping dimension=15, Similarity function=largest common subsequence	$172,492$ / $21,509$ / $270,165$	60 / 7.88 / 97
Algorithm 1	$p = 2.5$, $minsup = 8, 6, 4, 3, 2$, $min\_th = 0.1$	$10,011$	16

Table 6: Blocking and comparison stages on the synthetic dirty 10K dataset

The runtime results of the blocking and comparison phases for the synthetic dataset appear in Table 6, showing similar behavior as in Table 5. We observe that choosing an appropriate blocking key is critical not only in order to achieve high effectiveness, but also for generating a small number of comparisons. Again, Algorithm 1 is shown to be at the same scale in terms of combined runtime as the other blocking algorithms.

6. Related Work

Several surveys were published in the general area of entity resolution. Elmagarmid et al. provided a comprehensive survey covering the complete Deduplication process [28]. Christen's survey is dedicated to popular blocking methods, [3] including the suffix array blocking [22], the threshold-based canopy clustering [2], the $q\text{-}gram$ based indexing approach [24], the blocking approach [25], the sorted neighborhood method [23], and others. The survey reports on the effectiveness of these methods. Our empirical evaluation follows closely the empirical evaluation proposed there. It compares these blocking methods, over several settings. The datasets used there are publically available for download or can be generated by using FEBRL [15].

All blocking techniques described in this section require the definition of a blocking key, which is orthogonal to the selection of the blocking algorithm. A blocking key is a carefully chosen set of attributes whose values are used to separate tuples into blocks.

Blocking key composition requires knowledge of the distribution and error characteristics of the data, and is therefore hard to configure. Bilenko proposed a learning method to automatically select the best blocking key, using an approximation to the set cover problem [29]. Michelson and Knoblock [30] employ predicate-based formulations of learnable blocking functions for the same purpose. Our proposed algorithm, on the other hand, makes do with the need to configure a blocking key. Instead, the algorithm allows all attribute values to participate in the clustering criteria, in effect, using multiple keys instead of a single, global one.

Some works suggested the use of multiple passes of a blocking algorithm, each time with a different blocking key [16, 31, 20]. In these works, just like in the single pass algorithms, the blocking keys were designed by domain experts while the proposed algorithm relieves the user from this difficult task. In addition, the number of selected keys is usually small. In our proposed approach the algorithm chooses the best keys to be used from the set of all possible keys. Another main difference between the two approaches has to do with the relationships between blocks of different passes. Multipass algorithms performs independent runs for different blocking keys. As a result, a tuple may participate in many blocks, increasing the load for later stages of the process. In our approach, a tuple is restricted to participate in a limited number of comparisons using the neighborhood growth constraint.

All blocking techniques but Standard blocking [25] create overlapping blocks. This means that a tuple may belong to more than a single block and will ultimately be compared with all tuples which share a block with it. For example, Canopies [2] are overlapping blocks created by applying a cheap comparison metric to tuples. Canopy clusters are generated by randomly selecting a tuple from a pool, and adding close tuples to the cluster. The proposed algorithm also creates overlapping blocks using multiple passes, as discussed above.

Most blocking methods are implemented using an inverted index structure [3], where the blocking key values are transformed to a set of keys in the index structure, later used to extract blocks. In Q-gram indexing, also known as fuzzy blocking [15], the blocking key values are converted into a list of q-grams that are concatenated and used as keys in an inverted index. In suffix array based blocking [22] an inverted index is built from the blocking key values and their suffixes. The String-Map based indexing technique [27] builds a d-dimensional grid-based inverted index from a mapping of blocking key values to points in a multidimensional space. Using itemset mining, attribute-specific indexing or comparison is no longer needed.

Shu et al. propose a divisive hierarchical clustering algorithm that is based on spectral clustering [32]. After the clustering phase, neighboring clusters in a bipartition tree are searched for locating additional candidate tuple pairs. The algorithm performs slightly better than the Canopy Clustering method on relatively clean datasets and performs worse than the Canopy Clustering method on dirty datasets. We did not compare directly with this algorithm. However, all of our experiments demonstrate that the proposed algorithm is significantly superior to the Canopy Clustering algorithm on all types of datasets (real-world, synthetic, clean, and dirty).

Many of the blocking methods constrain the block size. This is done to avoid the cost of having to compare many candidate pairs whenever large blocks are generated. This constraint may be difficult to come-by and as opposed to our proposed approach

has no relation to the expected number of duplicates. To the best of our knowledge, our work is the first to utilize the expected number of duplicates, which in many scenarios is known in advance, to set a meaningful constraint on the number of comparisons.

Entity resolution can benefit from the use of semantic information. Yakout et al. [33] propose to use the entity's behavior against a data source by analyzing patterns for this entity in a transaction log. This additional data is an important source of information, especially when user-profile information is inaccurate or unavailable. Relational entity resolution [34] makes use of relational information between entities in order to enhance the effectiveness of the entity resolution process. In a blocking algorithm for such a setting, entities are placed in the same block, not only based on their similarity, but also due to their relationships. The proposed algorithm makes use of relational information, assigning different ids to identical q-grams, if their origin is in different attributes. As part of future work we plan to investigate additional ways to enhance our approach to deal with relational information as well.

Methods for improving the efficiency of the entity resolution process involve parallelization using Map Reduce [35, 36]. Kolb et al. [35] focus on strategies for similarity computation and classifier application on the cartesian product of two sources. Kirsten et al. [36] apply blocking as part of the partitioning strategies for parallelization. We intend to research parallelization as a tool for improving performance in future research.

The measures we used for comparing results to a golden standard are the popular recall, precision, and F-measure. Other measures also exist. For example, Menestrina et al. [37] propose the Generalized Merge Distance (GMD), using the operations of cluster splits and merges to measure distance among clusters.

7. Conclusions

We presented an effective and efficient blocking algorithm that eliminates the need to construct a complex blocking key and reduces the need for complex user-based tuning. The algorithm is based on mining maximal frequent itemsets. A tuple is broken into a sequence of items, where each item is part of an attribute value. The algorithm avoids performing attribute specific indexing or comparison, and therefore may perform well even in the absence of domain expert knowledge. The proposed approach discovers clusters of similar tuples under different sets of attributes (blocking keys) and therefore achieves higher recall and precision. We exhibited the effectiveness and efficiency of the algorithm on both real-world and synthetic datasets and compared it to other known blocking methods, showing it to be more effective.

In addition to block creation for the comparison and classification stages, the proposed algorithm provides insights as to the set of attributes that should be utilized in these stages. As such, we believe the algorithm benefit goes beyond that of a common blocking algorithm. We intend to investigate the impact of the algorithm outcome on the comparison and classification stages as part of our future work. In addition, we intend to investigate how to extend this approach to relational blocking. Finally, we also intend to explore how to extend the approach in scenarios where a crisp correspondence between the attributes is unavailable or when it is probabilistic [38].

Acknowledgments

The work was carried out in and partially supported by the Technion–Microsoft Electronic Commerce research center. We thank Peter Christen for useful comments.

References

[1] M. Hernandez, M. A. Hernàndez, S. Stolfo, Real-world data is dirty: Data cleansing and the merge/purge problem, Data Mining and Knowledge Discovery 2 (1998) 9–37.

[2] A. McCallum, K. Nigam, L. H. Ungar, Efficient clustering of high-dimensional data sets with application to reference matching, in: KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, 2000, pp. 169–178.

[3] P. Christen, A survey of indexing techniques for scalable record linkage and deduplication, IEEE Transactions on Knowledge and Data Engineering 99 (PrePrints). doi:http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.127.

[4] G. Grahne, J. Zhu, Fast algorithms for frequent itemset mining using fp-trees, IEEE Transactions on Knowledge and Data Engineering 17 (10) (2005) 1347–1362.

[5] L. Parsons, Evaluating subspace clustering algorithms, in: in Workshop on Clustering High Dimensional Data and its Applications, SIAM International Conference on Data Mining (SDM 2004), 2004, pp. 48–56.

[6] S. Chaudhuri, V. Ganti, R. Motwani, Robust identification of fuzzy duplicates, in: Proc. 21st Int. Conf. on Data Engineering, 2005, pp. 865–876.

[7] F. Naumann, M. Herschel, An Introduction to Duplicate Detection, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2010.

[8] R. Agrawal, T. Imielinski, A. N. Swami, Mining association rules between sets of items in large databases, in: P. Buneman, S. Jajodia (Eds.), Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993, ACM Press, 1993, pp. 207–216.

[9] D. I. Ignatov, S. O. Kuznetsov, Frequent itemset mining for clustering near duplicate web documents, in: 17th International Conference on Conceptual Structures, 2009, pp. 185–200.

[10] J. Han, M. Kamber, Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems), 1st Edition, Morgan Kaufmann, 2000.

[11] G. Yang, The complexity of mining maximal frequent itemsets and maximal frequent patterns, in: Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, ACM Press, 2004, pp. 344–353.

[12] D. Burdick, M. Calimlim, J. Gehrke, Mafia: A maximal frequent itemset algorithm for transactional databases, in: Proc. 17th Int. Conf. on Data Engineering, 2001, pp. 443–452.

[13] K. Gouda, M. J. Zaki, Efficiently mining maximal frequent itemsets, in: Proc. 2001 IEEE Int. Conf. on Data Mining, 2001, pp. 163–170.

[14] B. Goethals, M. J. Zaki, Advances in frequent itemset mining implementations: Report on fimi'03, in: SIGKDD Explorations, 2003, p. 2004.

[15] P. Christen, A. Pudjijono, Accurate synthetic generation of realistic personal information, in: PAKDD'09, Springer LNAI, Vol. 5476, Bangkok, Thailand, 2009, pp. 507–514.

[16] M. Weis, F. Naumann, Dogmatix tracks down duplicates in XML, in: SIGMOD'05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2005, pp. 431–442.

[17] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.

[18] W. E. Winkler, The state of record linkage and current research problems, Tech. rep., Statistical Research Division, U.S. Bureau of the Census (1999).

[19] G. Salton, A. Wong, C. S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620.

[20] U. Draisbach, F. Naumann, A generalization of blocking and windowing algorithms for duplicate detection, in: Proc. Int. Conf. on Data and Knowledge Engineering (ICDKE), 2011, pp. 18–24.

[21] M. Elfeky, V. Verykios, A. Elmagarmid, Tailor: A record linkage toolbox, in: Proc. 18th Int. Conf. on Data Engineering, 2002.

[22] A. Aizawa, K. Oyama, A fast linkage detection scheme for multi-source information integration, in: WIRI '05: Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration, IEEE Computer Society, Washington, DC, USA, 2005, pp. 30–39.

[23] M. A. Hernández, S. J. Stolfo, The merge/purge problem for large databases, SIGMOD Rec. 24 (2) (1995) 127–138.

[24] R. Baxter, P. Christen, T. Churches, A comparison of fast blocking methods for record linkage, in: KDD 2003 WORKSHOPS, 2003, pp. 25–27.

[25] I. P. Fellegi, A. B. Sunter, A theory for record linkage, Journal of the American Statistical Association 64 (328) (1969) 1183–1210.

[26] W. E. Winkler, Improved decision rules in the fellegi-sunter model of record linkage, in: Proceedings of the Section on Survey Research Methods, American Statistical Association, 1993, pp. 274–279.

[27] L. Jin, C. Li, S. Mehrotra, Efficient record linkage in large data sets, in: DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications, IEEE Computer Society, Washington, DC, USA, 2003, p. 137.

[28] A. K. Elmagarmid, P. G. Ipeirotis, V. S. Verykios, Duplicate record detection: A survey, IEEE Trans. Knowl. Data Eng. 19 (1) (2007) 1–16.

[29] M. Bilenko, Adaptive blocking: Learning to scale up record linkage, in: In Proceedings of the 6th IEEE International Conference on Data Mining (ICDM-2006), 2006, pp. 87–96.

[30] M. Michelson, C. A. Knoblock, Learning blocking schemes for record linkage, in: Proc. 21st National Conf. on Artificial Intelligence and 18th Innovative Applications of Artificial Intelligence Conf., 2006, pp. 440–445.

[31] W. Winkler, W. Yancey, E. Porter, Fast record linkage of very large files in support of decennial and administrative records projects, in: Proceedings of the Section on Survey Research Methods, American Statistical Association, 2010.

[32] L. Shu, A. Chen, M. Xiong, W. Meng, Efficient spectral neighborhood blocking for entity resolution, in: Proc. 27th Int. Conf. on Data Engineering, 2011, pp. 1067–1078.

[33] M. Yakout, A. K. Elmagarmid, H. Elmeleegy, M. Ouzzani, A. Qi, Behavior based record linkage, PVLDB 3 (1-2) (2010) 439–448.

[34] V. Rastogi, N. N. Dalvi, M. N. Garofalakis, Large-scale collective entity matching, PVLDB 4 (4) (2011) 208–218.

[35] L. Kolb, H. Köpcke, A. Thor, E. Rahm, Learning-based Entity Resolution with MapReduce, in: Proceedings of the third international workshop on Cloud data management, CloudDB '11, ACM, New York, NY, USA, 2011, pp. 1–6.

[36] T. Kirsten, L. Kolb, M. Hartung, A. Gross, H. Köpcke, E. Rahm, Data Partitioning for Parallel Entity Matching, in: 8th International Workshop on Quality in Databases, 2010.

[37] D. Menestrina, S. E. Whang, H. Garcia-Molina, Evaluating entity resolution results, PVLDB 3 (1-2) (2010) 208–219.

[38] X. L. Dong, A. Y. Halevy, C. Yu, Data integration with uncertainty, in: Proc. 33rd Int. Conf. on Very Large Data Bases, 2007, pp. 687–698.